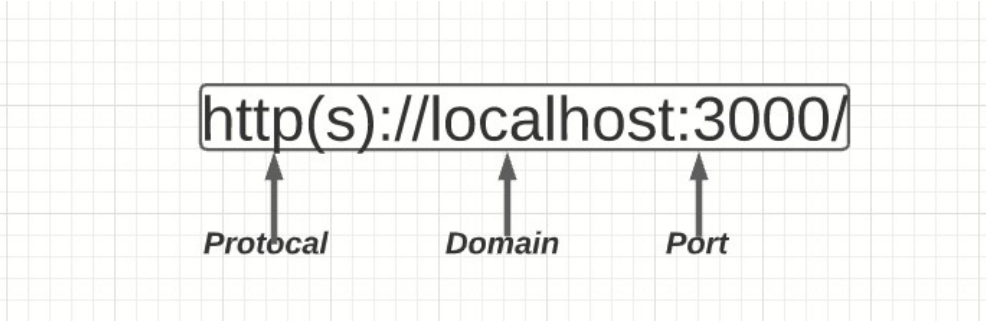
How to use CORS in Node.js with Express

section.io/engineering-education/how-to-use-cors-in-nodejs-with-express

Node.js is an open-source and cross-platform runtime used when executing JavaScript code on the server-side. One of the popular Node.js server frameworks is Express. Implementing CORS in Node.js helps you access numerous functionalities on the browser.

Express allows you to configure and manage an HTTP server to access resources from the same domain.

The three parts that form an origin are protocal, domain, and port.



Today, there are many applications that depend on APIs to access different resources. Some of the popular APIs include weather, time, and fonts. There are also servers that host these APIs and ensure that information is delivered to websites and other end points. Therefore, making cross-origin calls, is a popular use case for the modern web application.

Let's say accessing images, videos, iframes, or scripts from another server. This means that the website is accessing resources from a different origin or domain. When building an application to serve up these resources with Express, a request to such external origins may fail. This is where CORS comes in to handle cross-origin requests.

Goal

This guide will help you learn how to configure CORS with Express.

Prerequisites

To follow this article along, prior knowledge of Node.js and Express is essential.

What is CORS?

CORS stands for `Cross-Origin Resource Sharing`. It allows us to relax the security applied to an API. This is done by bypassing the `Access-Control-Allow-Origin` headers, which specify which `origins` can access the API.

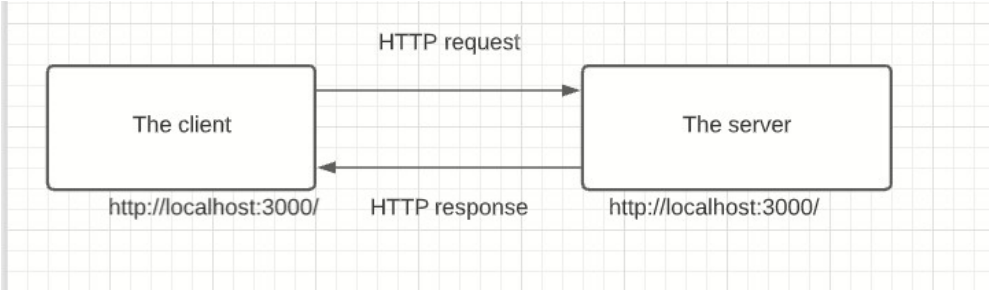
In other words, CORS is a browser security feature that restricts cross-origin HTTP requests with other servers and specifies which domains access your resources.

Check this guide to learn more about the CORS policy.

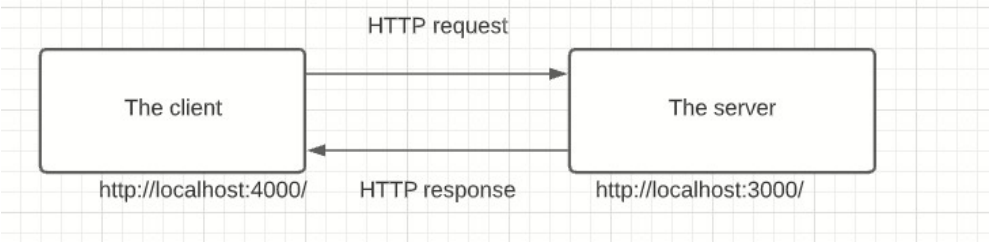
How CORS works

An API is a set procedure for two programs to communicate. This means that API resources are consumed by other clients and servers.

Here are two scenarios:



The client and the server have the same origin. In this example, accessing resources will be successful. You're trying to access resources on your server, and the same server handles the request.



The client and server have a different origin from each other, i.e., accessing resources from a different server. In this case, trying to make a request to a resource on the other server will fail.

This is a security concern for the browser. CORS comes into play to disable this mechanism and allow access to these resources. CORS will add a response header `access-control-allow-origins` and specify which origins are permitted. CORS ensures that we are sending the right headers.

Therefore, a public server handling a public API will add a CORS related header to the response. The browser on the client machine will look at this header and decide whether it is safe to deliver that response to the client or not.

Setting up CORS with Express

Let's create a very basic Express HTTP server endpoint that serves a GET response.

Make sure you have `Node.js runtime` installed and run `npm init -y` to start your simple express server project. To use CORS within a Node.js application, you need the `cor package` provided by the Node.js NPM registry. Go ahead and install CORS alongside the following other packages using the below command.

```
npm i cors express nodemon
```

Creating a simple Express GET request

Below is a simple `index.js` express server.

```
const express = require('express');
const app = express();

    const ingredients = [
    {
        "id": "1",
        "item": "Bacon"
    },
    {
        "id": "2",
        "item": "Eggs"
    },
    {
        "id": "3",
        "item": "Milk"
    },
    {
        "id": "4",
        "item": "Butter"
    }
];

app.get('/ingredients', (req, res) =>{
    res.send(ingredients);
});
app.listen(6069);
```

The code above depicts a simple HTTP server using Express. Check this guide and learn how to create one.

Run the server with `npm nodemon`. Navigate to `http://localhost:6069/ingredients` on your browser. You will be served with these ingredients text items.

In this example, cross-origin is allowed because you're currently on the same domain, and you are executing this request from the same domain.

Let's now try to get the ingredients using the fetch command.

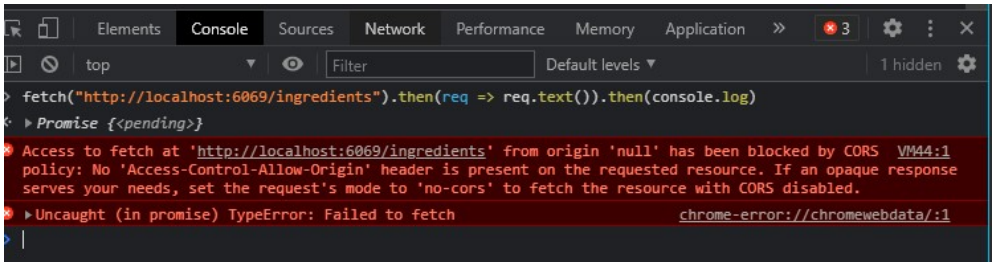
I am going to execute that same request but from another site instead. In this case, I used `https://www.section.io`.

Open `https://www.section.io` on your browser and execute the following fetch request from the browser's console using a Fetch API.

```
fetch("http://localhost:6069/ingredients").then(req => req.text()).then(console.log)
```

Make sure the server is up and running before performing the request above.

We are fetching the ingredients information from another origin domain. The origin of this URL is not the one allowed to receive this response from this server. Therefore, it will throw the below CORS error.



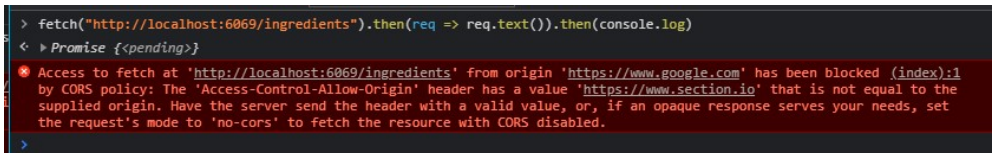
To solve this error, we need to add the CORS header to the server and give `https://www.section.io` access to the server response.

Include the following in your `index.js` file.

```
const cors = require('cors');
app.use(cors({
    origin: 'https://www.section.io'
}));
```

If you now perform the fetch command, it should work fine.

However, if a fetch request is made from another web page, it will fail and throw the following error.



This means that you only have access to our server's resources. You can have an array of these multiple origins, as shown below.

```
const cors = require('cors');
app.use(cors({
    origin: ['https://www.section.io', 'https://www.google.com/']
}));
```

Nevertheless, the API can be public, and any cross-origin APIs and servers can access these resources. The code block below will ensure any page can access the ingredient resources.

```
app.use(cors({
    origin: '*'
}));
```

The Asterisk symbol will create the CORS header, and any origin can, therefore, get the response of this localhost server.

Since a specific origin is not defined here, `app.use(cors())` will also get this done.

You can also have dynamic origins. These are whitelisted origins that have access to your API. This could be used to pull resources from a database.

Let's say you have different accounts, i.e. developer accounts, and you might want them to access the API. To have this dynamic whitelisting, use this origin function which returns these domains individually.

```
const whitelist = ['http://developer1.com', 'http://developer2.com']
const corsOptions = {
  origin: (origin, callback) => {
    if (whitelist.indexOf(origin) !== -1) {
      callback(null, true)
    } else {
      callback(new Error())
    }
  }
}
```

Performing more requests

An express server has many specific ways of managing CORS to determine what other servers and APIs can be accessed.

For example, a server is comprised of several methods. However, not every method should be exposed to other origins. Thus, within CORS middleware, you can specify which methods can be accessed by the CORS policy.

These methods include `GET`, `POST`, `DELETE`, `UPDATE`, and `OPTIONS`.

```
app.use(cors({
    methods: ['GET','POST','DELETE','UPDATE','PUT','PATCH']
}));
```

You can even specify which routes of your server can be accessed.

```
app.get('/ingredients', cors(), (req, res, next) => {
    res.send(ingredients);
});
```

Use case

Let's say you're using React to build a front-end application. Eventually, you'll be connecting it to a back-end API. The app might run into an issue if CORS is not set up. Simply because both the front end and the back end are from different origins from each other.

CORS goes hand in hand with APIs. A good use case scenario of CORS is when developing RESTful APIs. For example, creating a Node.js RESTful API, similar to this RESTful Web API in Node.js using PostgresSQL and Express.

Conclusion

When you deploy an application on the server, you should not accept requests from every domain. Instead, you should specify which origin can make requests to your server.

This way, you are able to block users who attempt to clone your site or make requests from an unauthorized servers. This is important an security measure. Check this CORS NPM registry and learn more on where to use CORS in your Express application.